

27장 배열 메서드

```
// [1, 2, 3, 4]의 모든 요소의 누적을 구한다.  
const sum = [1, 2, 3, 4].reduce((accumulator, currentValue,  
index, array) => accumulator + currentValue, 0);  
  
console.log(sum); // 10
```

배열 메서드

- 원본 배열을 직접 변경하는 메서드와
- 원본 배열을 직접 변경하지 않고 새로운 배열을 생성하여 반환하는 메서드가 있다

Array.isArray

- 전달된 인수가 배열이면 `true`, 배열이 아니면 `false` 를 반환

```
// true  
Array.isArray([]);  
Array.isArray([1, 2]);  
Array.isArray(new Array());
```

Array.prototype.indexOf

- 원본 배열에서 인수로 전달된 요소를 검색하여 인덱스를 반환
- 원본 배열에 인수로 전달한 요소가 없다면 -1을 반환

```
const Log = (res) => console.log  
const arr = [1, 2, 2, 3];  
  
Log(arr.indexOf(2));  
Log(arr.indexOf(2));
```

```
const Log = (res) => console.log(res);  
const arr = [1, 2, 2, 3];  
  
Log(arr.indexOf(2));  
Log(arr.indexOf(2));  
//2번째부터 검색 시작, 비울 경우 0부터 시작  
Log(arr.indexOf(2, 2));  
  
1  
1  
2
```

```
//2번째부터 검색 시작 , 비울 경우 0부터
Log(arr.indexOf(2, 2));
```

Array.prototype.push

- 인수로 전달받은 모든 값을 **원본 배열의 마지막 요소**로 추가
- 변경된 length 프로퍼티 값을 반환
- **원본 배열을 직접 변경한다.**

```
const Log = (res) => console.log
const arr = [1, 2];

let result = arr.push(3, 4);
Log(result); // 4
Log(arr); // [1,2,3,4]
```

```
> const Log = (res) => console.log(res);
const arr = [1, 2];

let result = arr.push(3, 4);
Log(result); // 4

Log(arr); // [1,2,3,4]
4
▶ (4) [1, 2, 3, 4]
```

- 다만 push메서드는 성능에 좋지 않으므로 length 프로퍼티를 사용하여 직접 추가하는 것이 좋다

```
const Log = (res) => console.log
const arr = [1, 2];

arr[arr.length] = 3;
Log(arr);
```

```
> const Log = (res) => console.log(res);
const arr = [1, 2];

arr[arr.length] = 3;
Log(arr);

▶ (3) [1, 2, 3]
```

Array.prototype.pop

- 원본 배열에서 **마지막 요소를 제거하고 제거한 요소를 반환**한다.
- 원본 배열이 빈 배열이면 undefined를 반환한다.
- **원본 배열을 직접 변경**한다.

```
const Log = (res) => console
const arr = [1, 2];

let result = arr.pop();
Log(result);
Log(arr);
```

```
const Log = (res) => console.log(res);
const arr = [1, 2];

let result = arr.pop();
Log(result);
Log(arr);

2
▶ [1]
```

pop , push 메서드를 이용한 stack 구현

stack

스택은 데이터를 마지막에 밀어 넣고, 마지막에 밀어 넣은 데이터를 먼저 꺼내는

후입 선출(LIFO - Last In First Out) 방식의 자료구조

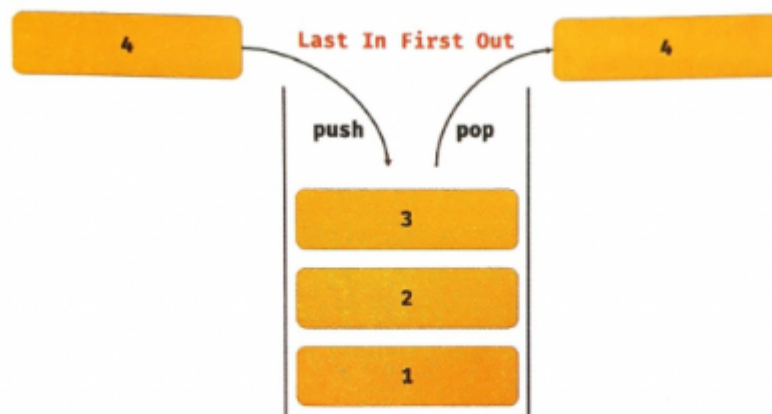


그림 27-3 스택

https://velog.io/@april_5/모던-자바스크립트-Deep-Dive-배열#-2785-arrayprototypeunshift

이점

1. 메모리 관리

- 함수 호출이 끝나면 스택 프레임이 팝 되면서 메모리를 자동으로 해제 하여 메모리 관리

2. undo 기능 구현

- 텍스트 에디터나 그래픽 편집기에서 이전 작업을 되돌리는 Undo 기능을 구현
- 작업을 스택에 푸시하고, Undo 시 스택에서 팝하여 이전 상태로 복원

3. 작업 순서 관리

- 특정 작업 순서를 유지
- ex) 실행 컨텍스트 스택

```
class Stack {
  #array; // private class member, 외부에서 접근 불가

  constructor() {
    this.#array = [];
  }

  // 스택의 가장 마지막에 데이터를 밀어 넣는다.
  addToArray(item) {
    this.#array.push(item);
  }

  // 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신 데이터를 꺼내
  popToArray() {
    return this.#array.pop();
  }

  // get 메서드를 사용하여
  // 스택의 복사본 배열을 반환한다.
  get array() {
    return [...this.#array];
  }

  // set 메서드를 사용하여
  // 배열 초기값을 설정한다.
  set array(arr) {
    if (Array.isArray(arr)) {
      this.#array = arr;
    }
  }
}
```

```

    } else {
      throw new TypeError('array 필드에는 배열만 설정할 수 있습니다.
    }
  }
}

const Log = (res) => console.log(res);
const errLog = (res) => console.error(res);
const stack = new Stack(); // stack 인스턴스 생성

//try {
//  Log(stack.#array); // private 접근 불가 확인 - SyntaxError
//} catch (e) {
//  errLog(e.message);
//  SyntaxError: Private field '#array' must be declared in an
//}

// get 메서드 작동 확인
Log(stack.array); // []

//try {
//  stack.array = 12; // 배열이 아닌 값 설정 시 - TypeError
//} catch (e) {
//  errLog(e.message); // 'array 필드에는 배열만 설정할 수 있습니다.
//}

stack.array = [1, 2]; // 배열값 설정 시
Log(stack.array); // [1, 2]

stack.addToArray(3); // push
Log(stack.array); // [1, 2, 3]

stack.popToArray(); // pop
Log(stack.array); // [1, 2]

```

Array.prototype.unshift

- 인수로 전달받은 모든 값을 **원본 배열의 선두에 요소로 추가**하고
- 변경된 length 프로퍼티 값을 반환한다.
- **원본 배열을 직접 변경**

```
const Log = (res) => console.log
const arr = [1, 2];

//인수로 전달받은 모든 값을
//원본 배열의 선두에 요소로 추가하고
//변경된 length 값을 반환한다.
let result = arr.unshift(3, 4);
Log(result); // 4

//원본 배열을 직접 변경한다.
Log(arr); // [3, 4, 1, 2]
```

```
const Log = (res) => console.log(res);
const arr = [1, 2];

//인수로 전달받은 모든 값을
//원본 배열의 선두에 요소로 추가하고
//변경된 length 값을 반환한다.
let result = arr.unshift(3, 4);
Log(result); // 4

//원본 배열을 직접 변경한다.
Log(arr); // [3, 4, 1, 2]

4
▶ (4) [3, 4, 1, 2]
```

Array.prototype.shift

- 배열에서 **첫 번째 요소를 제거**하고 제거한 요소를 반환한다.
- **원본 배열을 직접 변경**

```
const Log = (res) => console.log
const arr = [1, 2];

let result = arr.shift();
Log(result); // 1

Log (arr); // [2]
```

```
const arr = [1, 2];

let result = arr.shift();
Log(result); // 1

Log (arr); // [2]

1
▶ [2]
```

shift, push 메서드를 이용한 queue 구현

queue

데이터를 마지막에 밀어 넣고, 가장 먼저 밀어 넣은 데이터를 먼저 꺼내는 선입 선출(**FIFO** - **First in First Out**) 방식의 자료구조



https://velog.io/@april_5/모던-자바스크립트-Deep-Dive-배열#-2785-arrayprototypeunshift

이점

1. 순서보장
2. 데이터 스트림 처리
 - 실시간으로 들어오는 데이터를 처리하고 저장하는 데 유용
 - 주로 데이터가 생성되고 소비되는 속도가 다를 때 사용
3. 동시성 제어
 - 생산자-소비자 문제 해결에 사용



생산자는 데이터를 만들어 버퍼에 저장하고(채워나가고), 소비자는 버퍼에 있는 데이터를 꺼내 소비하는(비우는) 프로세스 버퍼는 공유 자원이므로 버퍼에 대한 접근 즉, 저장하고 꺼내는 일들이 상호 배제되어야 함

```

class Queue {
  #array; // private class member, 외부에서 접근 불가

  constructor() {
    this.#array = [];
  }

  // 큐의 가장 마지막에 데이터를 밀어 넣는다.
  enqueue(item) {
    this.#array.push(item);
  }

  // 큐의 가장 처음 데이터를 꺼낸다
  dequeue() {
    return this.#array.shift();
  }

  // get 메서드를 사용하여
  // 스택의 복사본 배열을 반환한다.
  get array() {
    return [...this.#array];
  }

  // set 메서드를 사용하여
  // 배열 초기값을 설정한다.
  set array(arr) {
    if (Array.isArray(arr)) {
      this.#array = arr;
    } else {
      throw new TypeError('array 필드에는 배열만 설정할 수 있습니다.
    }
  }
}

const Log = (res) => console.log(res);
const errLog = (res) => console.error(res);
const queue = new Queue(); // queue 인스턴스 생성

```



```

queue.array = [1, 2]; // 배열값 설정 시
Log(queue.array); // [1, 2]

queue.enqueue(3); // enqueue
Log(queue.array); // [1, 2, 3]

Log(queue.dequeue()); // dequeue , 1
Log(queue.array); // [2 , 3]

```

Array.prototype.concat

- 원본 배열의 마지막 요소로 추가한 새로운 배열을 반환
- 원본 배열은 변경되지 않는다.

```

const Log = (res) => console.log(res);
const arr1 = [1, 2];
const arr2 = [3, 4];

// 배열 arr2를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환
// 인수로 전달한 값이 배열인 경우 배열을 해체하여 새로운 배열의 요소로 추가
let result = arr1.concat(arr2);
Log(result); // [1, 2, 3, 4]

// 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환한다.
result = arr1.concat(3);
Log(result); // [1, 2, 3]

// 배열 arr2와 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을
result = arr1.concat(arr2, 5);
Log(result); // [1, 2, 3, 4, 5]

```

```
// 배열 arr2와 숫자를 원본 배열 arr1에 추가
result = arr1.concat(arr2, 5);
Log(result); // [1, 2, 3, 4, 5]

▶ (4) [1, 2, 3, 4]
▶ (3) [1, 2, 3]
▶ (5) [1, 2, 3, 4, 5]
```

Array.prototype.splice

- 원본 배열 **중간에** 요소를 추가하거나 **중간에 있는** 요소를 제거하는 경우 **splice** 메서드를 사용한다
- 사용 인수
 - **Array.prototype.splice(start , deleteCount , items)**
 - **start**: 원본 배열의 요소를 제거하기 시작할 인덱스
 - **deleteCount** (옵션): 원본 배열의 요소를 제거하기 시작할 인덱스인 **start** 부터 제거할 요소의 개수
 - **items** (옵션): 제거한 위치에 삽입할 요소들의 목록
- 원본 배열을 직접 변경한다

```
const Log = (res) => console.log(res);
const arr = [1, 2, 3, 4];

// 원본 배열의 인덱스 1부터 2개의 요소를 제거하고
// 그 자리에 새로운 요소 20, 30을 삽입한다.
const result_1 = arr.splice(1, 2, 20, 30);
Log(arr)

// 원본 배열의 인덱스 1부터 0개의 요소를 제거하고
// 그 자리에 새로운 요소 100을 삽입한다.
const result_2 = arr.splice(1, 0, 100);
Log(arr)

// 원본 배열의 인덱스 1부터 2개의 요소를 제거한다.
const result_3 = arr.splice(1, 2);
```

```
Log(arr)
```

```
// 원본 배열의 인덱스 1부터 모든 요소를 제거한다.
```

```
const result_4 = arr.splice(1);
```

```
Log(arr)
```

```
▶ (4) [1, 20, 30, 4]
▶ (5) [1, 100, 20, 30, 4]
▶ (3) [1, 30, 4]
▶ [1]
```

Array.prototype.slice

- 인수로 전달된 범위의 요소들을 복사하여 배열로 반환
- 사용 인수
 - **Array.prototype.slice(start, end)**
 - **start** : 복사를 시작할 인덱스
 - **end** : 복사를 종료할 인덱스
- 원본 배열은 변경되지 않는다.

```
const Log = (res) => console.log(res);
```

```
const arr = [1, 2, 3];
```

```
// arr[0]부터 arr[1] 이전(arr[1] 미포함)까지 복사하여 반환한다.
```

```
Log(arr.slice(0, 1)); // → [1]
```

```
// arr[1]부터 arr[2] 이전(arr[2] 미포함)까지 복사하여 반환한다.
```

```
Log(arr.slice(1, 2)); // → [2]
```

```
// 배열의 끝에서부터 요소를 한 개 복사하여 반환한다.
```

```
Log(arr.slice(-1)); // → [3]
```

```
// 배열의 끝에서부터 요소를 두 개 복사하여 반환한다.
Log(arr.slice(-2)); // → [2, 3]

// 원본은 변경되지 않는다.
Log(arr); // [1, 2, 3]
```

```
▶ [1]
▶ [2]
▶ [3]
▶ (2) [2, 3]
▶ (3) [1, 2, 3]
```

- `slice` 메서드의 인수를 모두 생략하면 얕은 복사를 통해 생성된 복사본을 반환

```
const Log = (res) => console.log(res);
const arr = [1, 2, 3];

// 인수를 모두 생략하면 원본 배열의 복사본 (얕은 복사)을 생성하여 반환한다
const copy = arr.slice();
Log(copy); // [1, 2, 3]
Log(copy === arr); // false
```

```
▶ (3) [1, 2, 3]
false
```

Array.prototype.join

- 원본 배열의 모든 요소를 문자열로 변환
- 전달받은 구분자로 연결한 문자열을 반환
- 구분자는 생략 가능하며 기본 구분자는 콤마(,)

```
const Log = (res) => console.log(res);
const arr = [1, 2, 3, 4];
```

```
// 기본 구분자는 ', '이다.
// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 기본 구분자 ', '로 연결
Log(arr.join()); // → '1,2,3,4';

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 빈문자열로 연결한 문자열
Log(arr.join('')); // → '1234'

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 구분자 ':'로 연결한 문자열
Log(arr.join(':')); // → '1:2:3:4'
```

```
Log(arr.join(':')); // → '1:2:3:4'
1,2,3,4
1234
1:2:3:4
```

Array.prototype.reverse

- 배열의 순서를 반대로 뒤집는다.
- 변경된 배열을 반환한다.
- 원본 배열은 변경된다.

```
const Log = (res) => console.log(res);
const arr = [1, 2, 3];
const result = arr.reverse();

// reverse 메서드는 원본 배열을 직접 변경한다.
Log(arr); // [3, 2, 1]
// 반환값은 변경된 배열이다.
Log(result); // [3, 2, 1]
```

```
▶ (3) [3, 2, 1]
▶ (3) [3, 2, 1]
```

Array.prototype.fill

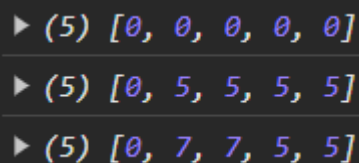
- 인수로 전달받은 값을 배열의 처음부터 끝까지 요소로 채운다.
- 원본 배열은 변경된다.

```
const Log = (res) => console.log(res);
const arr = [1, 2, 3, 4, 5];

// 인수로 전달 받은 값 0을 배열의 처음부터 끝까지 요소로 채운다.
arr.fill(0);
Log(arr); // [0, 0, 0, 0, 0]

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 끝까지 요소로 채운다.
arr.fill(5, 1);
Log(arr); // [0, 5, 5, 5, 5]

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 3 이전(인덱스 3 미포함)까지
arr.fill(7, 1, 3);
Log(arr); // [0, 7, 7, 5, 5]
```



```
▶ (5) [0, 0, 0, 0, 0]
▶ (5) [0, 5, 5, 5, 5]
▶ (5) [0, 7, 7, 5, 5]
```

- 배열을 생성하면서 특정값으로 변경 가능

```
const Log = (res) => console.log(res);
const arr = new Array(3);
//배열의 특정 인덱스에 요소가 없을 때, 해당 인덱스는 빈 슬롯(empty slot)
Log(arr); // [empty × 3]

// 인수로 전달받은 값 1을 배열의 처음부터 끝까지 요소로 채운다.
const result = arr.fill(1);

// fill 메서드는 원본 배열을 직접 변경한다.
```

```
Log(arr); // [1, 1, 1]
```

// fill 메서드는 변경된 원본 배열을 반환한다.

```
Log(result); // [1, 1, 1]
```

```
▶ (3) [empty × 3]
```

```
▶ (3) [1, 1, 1]
```

```
▶ (3) [1, 1, 1]
```

Array.prototype.includes

- 배열 내에 특정 요소가 포함되어 있는지 확인하여 `true` 또는 `false` 를 반환
- 사용 인수
 - `Array.prototype.includes(searchElement , fromIndex)`
 - `searchElement` : 검색할 대상 지정
 - `fromIndex` : 검색할 인덱스 지정

```
const Log = (res) => console.log(res);  
const arr = [1, 2, 3];
```

// 배열에 요소 1이 포함되어 있는지 인덱스 1부터 확인한다.

```
Log(arr.includes(1, 1)); // → false
```

// 배열에 요소 3이 포함되어 있는지 인덱스 2(arr.length - 1)부터 확인한다.

```
Log(arr.includes(3, -1)); // → true
```

```
Log(arr.includes(3, -1)); // → true
```

```
false
```

```
true
```

Array.prototype.flat

- 인수로 전달한 깊이 만큼 재귀적으로 배열을 평탄화
- 인수를 생략할 경우 기본값은 1이다.
 - 인수로 `Infinity`를 전달하면 중첩 배열 모두를 평탄화

```
const Log = (res) => console.log(res);

Log([1, [2, 3, 4, 5]].flat()); // → [1, 2, 3, 4, 5]

//깊이 값의 기본값은 1이다.
Log([1, [2, [3, [4]]]].flat()); // → [1, 2, [3, [4]]]
Log([1, [2, [3, [4]]]].flat(1)); // → [1, 2, [3, [4]]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 2로 지정하여 2단계 깊이까지 평
탄화한다.
Log([1, [2, [3, [4]]]].flat(2)); // → [1, 2, 3, [4]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 Infinity로 지정하여 중첩 배열
모두를 평탄화한다.
Log([1, [2, [3, [4]]]].flat(Infinity)); // → [1, 2, 3, 4]
```

배열 고차함수

고차 함수는 외부 상태의 변경이나 가변 데이터를 피하고 **불변성**을 지향하는

함수형 프로그래밍에 기반을 두고 있다.

최종적으로

순수 함수를 통해 부수 효과를 최대한 억제하여 오류를 피하고 프로그램의 안정성을 높이려
는 노력의 일환이라 볼 수 있다.

Array.prototype.sort

- 배열의 요소를 정렬
- 오름차순으로 정렬

- 원본 배열을 직접 변경하며 정렬된 배열을 반환한다.

매개변수

```
sort(compareFn)
```

- compareFn

정렬 순서를 정의하는 함수. 생략하면 배열은 각 요소의 문자열 변환에 따라 각 문자의 유니코드 코드 포인트 값에 따라 정렬됩니다.

사용

```
const Log = (res) => console.log(res);
const alpha = ['A', 'C', 'F', 'A'];

// 오름차순(ascending) 정렬
Log(alpha.sort());

// sort 메서드는 원본 배열을 직접 변경한다.
Log(alpha); //
```

```
// sort 메서드는 원본 배열을 직접 변경한다.
Log(alpha); //
▶ (4) ['A', 'A', 'C', 'F']
▶ (4) ['A', 'A', 'C', 'F']
```

- 숫자 요소 배열을 정렬시엔 유니코드 포인트의 순서를 따른다.

```
const Log = (res) => console.log(res);
const points = [123, 56325, 5135, 5325];

// 숫자 배열의 오름차순 정렬. 비교 함수의 반환값이 0보다 작으면
// a를 우선하여 정렬한다.
// 고차함수 이므로 콜백함수를 매개변수로 받는다
```

```
Log(points.sort((a, b) => a - b));
```

```
// 숫자 배열에서 최소/최대값 취득
```

```
Log(points[0], points[points.length]);
```

```
Log(points[0], points[points.length]);  
▶ (4) [123, 5135, 5325, 56325]  
123
```

- 비 아스키 문자 정렬시
- 악센트 부호가있는 문자 (e, é, è, a, ä 등)가있는 문자열을 정렬 시
- String의 localeCompare() 사용

```
const Log = (res) => console.log(res);  
const items = ["réservé", "premier", "cliché", "communiqué",  
  .sort((a, b) => {  
    return a.localeCompare(b);  
  }));  
Log(items)
```

```
▶ (6) ['adieu', 'café', 'cliché', 'communiqué', 'premier', 'réservé']
```

Array.prototype.forEach

- for 문 대체 고차 함수
- 자신이 호출한 배열을 순회하면서 처리를 콜백함수로 전달받아 반복 호출

매개변수

```
forEach(callbackFn, thisArg)
```

callbackFn

- 배열의 각 요소에 대해 실행할 함수

```
forEach(callbackFn(element, index, array) => {})
```

- element : 처리 중인 현재 요소
- index : 처리 중인 현재 요소의 인덱스
- array : forEach()를 호출한 배열

thisArg

- callback의 this 바인딩 용

```
const Log = (res) => console.log(res);
const arr = [0, 1];
const obj1 = {name: "kim"};
const obj2 = {name: "park"};

arr.forEach(function(element){
  Log(`${this.name} - ${element}`);
});

arr.forEach(function(element){
  Log(`${this.name} - ${element}`);
}, obj1);

arr.forEach(function(element){
  Log(`${this.name} - ${element}`);
}, obj2);

arr.forEach((element) => {
  Log(`${this.name} - ${element}`);
});
```

```
- 0
- 1
kim - 0
kim - 1
park - 0
park - 1
- 0
- 1
- 0
- 1
- 0
- 1
```

```
arr.forEach((element) => {
    Log(`${this.name} - ${element}`);
}, obj1);

arr.forEach((element) => {
    Log(`${this.name} - ${element}`);
}, obj2);
```

- 활용

```
const Log = (res) => console.log(res);
class Counter {
    constructor() {
        this.sum = 0;
        this.count = 0;
    }
    add(array) {
        // 오직 함수 표현식만 자신의 this 바인딩을 가집니다.
        array.forEach(function countEntry(entry) {
            this.sum += entry;
            ++this.count;
        }, this);
    }
}

const obj = new Counter();
obj.add([2, 5, 9]);
Log(obj.count); // 3
Log(obj.sum); // 16
```



```
3
16
```

- 만약 위의 예제에서 thisarg 인 this를 뺀다면?

```

▶ Uncaught TypeError: Cannot read properties of undefined (reading 'sum')
  at countEntry (<anonymous>:10:7)
  at Array.forEach (<anonymous>)
  at Counter.add (<anonymous>:9:11)
  at <anonymous>:17:5

```

strict mode 안의 일반함수로 정의된 함수의 this는 **undefined** 이기 때문에 sum이 존재하지 않는다

활용

```

const Log = (res) => console.log(res);
const numbers = [1, 2, 3];
let pows = [];

numbers.forEach(item => pows.push(item ** 2));
Log(pows); // [1, 4, 9]

[1, 2].forEach((item, index, arr) => {
  Log(`요소값: ${item}, 인덱스: ${index}, arr: ${JSON.stringify(arr)}`);
});

[1, 2].forEach(function (item, index, arr) {
  Log(`요소값: ${item}, 인덱스: ${index}, arr: ${JSON.stringify(arr)}`);
});

```

```

▶ (3) [1, 4, 9]

```

```

요소값: 1, 인덱스: 0, arr: [1,2] , this: [object Window]

```

```

요소값: 2, 인덱스: 1, arr: [1,2] , this: [object Window]

```

```

요소값: 1, 인덱스: 0, arr: [1,2] , this: [object Window]

```

```

요소값: 2, 인덱스: 1, arr: [1,2] , this: [object Window]

```

Array.prototype.map

- 자신을 호출한 배열의 모든 요소를 순회하면서 인수로 전달받은 콜백 함수를 반복 호출
- **콜백 함수의 반환값들로 구성된 새로운 배열을 반환**
- 원본 배열은 변경되지 않는다.

매개변수

```
arr.map(callback(currentValue[, index[, array]]), thisArg)
```

- callback
 - currentValue : 처리할 현재 요소
 - index : 처리할 현재 요소의 인덱스
 - array : map() 을 호출한 배열
- thisArg
 - forEach와 동일

활용

```
const Log = (res) => console.log(res);
const numbers = [1, 4, 9];

// map 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함수를 반복 호출
// 그리고 콜백 함수의 반환값들로 구성된 새로운 배열을 반환한다.
const roots = numbers.map(item => Math.sqrt(item));

Log(roots);    // [ 1, 2, 3 ]
Log(numbers);  // [ 1, 4, 9 ]
```

```
▶ (3) [1, 2, 3]
▶ (3) [1, 4, 9]
```

length

- map 메서드가 생성하여 반환하는 새로운 배열의 length 프로퍼티 값은 map 메서드를 호출한 배열의 length 프로퍼티 값과 반드시 일치한다.
- map 메서드를 호출한 배열과 map 메서드가 생성하여 반환한 배열은 1:1 매핑

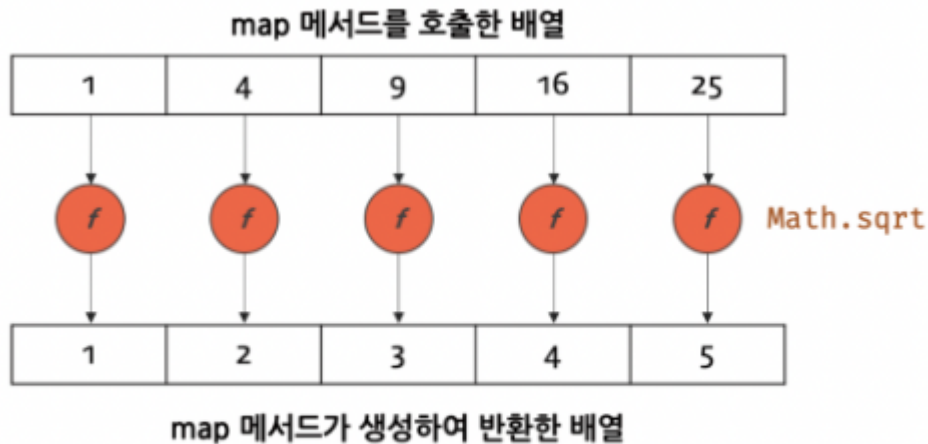


그림 27-9 Array.prototype.map

<https://velog.io/@jjinichoi/모던-자바스크립트-Deep-Dive-27장-배열>

Array.prototype.filter

- filter 메서드는 자신을 호출한 배열의 모든 요소를 순회하면서 인수로 전달받은 콜백 함수를 반복 호출
- 콜백 함수의 반환 값이 true인 요소로만 구성된 새로운 배열을 반환
- 원본 배열은 변경되지 않는다.

매개변수

```
filter(callbackFn, thisArg)
```

- callbackFn
 - 배열의 각 요소에 대해 실행할 함수
 - element : 처리 중인 현재 요소
 - index : 처리 중인 현재 요소의 인덱스

- array : filter()를 호출한 배열
- thisArg
 - forEach와 동일

활용

```
const Log = (res) => console.log(res);
function isBigEnough(value) {
  //10보다 작은 값은 걸러짐
  return value >= 10;
}

const filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
Log(filtered)
```

▶ (3) [12, 130, 44]

```
const Log = (res) => console.log(res);
const fruits = ["apple", "banana", "grapes", "mango", "orange"]

/**
 * 검색 조건에 따른 배열 필터링(쿼리)
 */
function filterItems(arr, query) {
  return arr.filter((el) => el.toLowerCase().includes(query.t
}

Log(filterItems(fruits, "ap")); // ['apple', 'grapes']
Log(filterItems(fruits, "an")); // ['banana', 'mango', 'orange']
```

▶ (2) ['apple', 'grapes']

▶ (3) ['banana', 'mango', 'orange']

length

filter 메서드가 생성하여 반환한 새로운 배열의 length 프로퍼티 값은 filter 메서드를 호출한 배열의 length 프로퍼티 값과 같거나 작다.

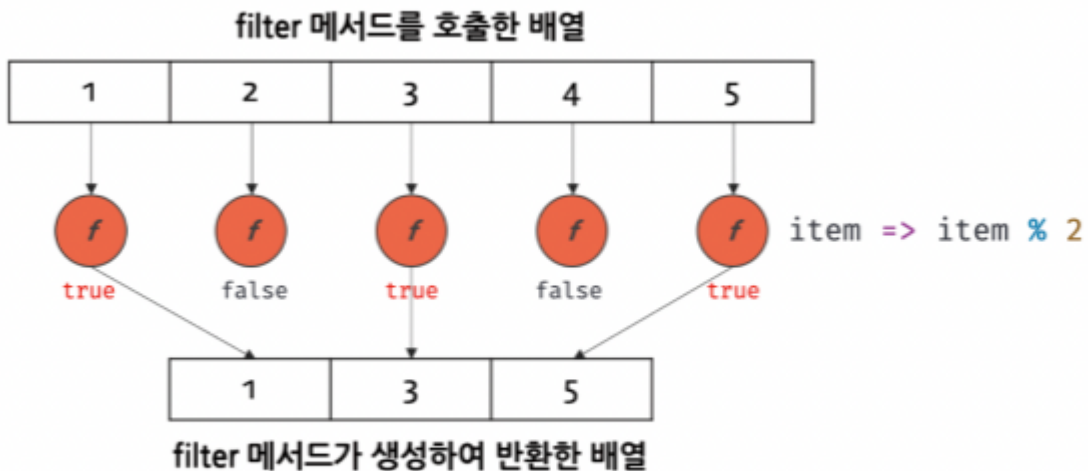


그림 27-10 Array.prototype.filter

<https://velog.io/@jjinichoi/모던-자바스크립트-Deep-Dive-27장-배열>

Array.prototype.reduce

- 자신을 호출한 배열의 모든 요소를 순회하면서 인수로 전달받은 콜백 함수를 반복 호출
- 콜백 함수의 반환값을 다음 순회 시에 콜백 함수의 첫 번째 인수로 전달하면서 콜백 함수를 호출하여 하나의 결과값을 만들어 반환
- 원본 배열은 변경되지 않는다.

매개변수

```
arr.reduce(callback[, initialValue])  
arr.reduce(callback((accumulator, currentValue, currentIndex, ar
```

- callback
 - 배열의 각 요소에 대해 실행할 함수

- accumulator
 - 콜백의 반환값을 누적
 - 콜백의 이전 반환값 혹은
 - 콜백의 첫 번째 호출이면서 `initialValue` 를 제공한 경우에는 `initialValue` 의 값
- currentValue
 - 처리할 현재 요소
- currentIndex
 - 처리할 현재요소의 인덱스
 - initialValue를 제공한 경우엔 0 , 아니면 1
- array
 - `reduce()` 를 호출한 배열
- initialValue
 - 최초 호출에서 첫 번째 인수에 제공하는 값
 - 초기값을 제공하지 않으면 배열의 첫 번째 요소를 사용
- `reduce` 메서드를 호출할 때는 **언제나 초기값을 지정하는 것이 안전**

활용

```
const Log = (res) => console.log(res);
// [1, 2, 3, 4]의 모든 요소의 누적을 구한다.
const sum = [1, 2, 3, 4].reduce((accumulator, currentValue,
index, array) => {
  Log(`accumulator : ${accumulator} currentValue : ${currentValue} index : ${index} array : ${array}`)
  return accumulator + currentValue
}, 0);

Log(sum); // 10
```

```

accumulator : 0 currentValue : 1 index : 0 array : 1,2,3,4
accumulator : 1 currentValue : 2 index : 1 array : 1,2,3,4
accumulator : 3 currentValue : 3 index : 2 array : 1,2,3,4
accumulator : 6 currentValue : 4 index : 3 array : 1,2,3,4
10

```

구분	콜백 함수에 전달되는 인수				콜백 함수의 반환값
	accumulator	currentValue	index	array	
첫 번째 순회	0 (초기값)	1	0	[1, 2, 3, 4]	1 (accumulator + currentValue)
두 번째 순회	1	2	1	[1, 2, 3, 4]	3 (accumulator + currentValue)
세 번째 순회	3	3	2	[1, 2, 3, 4]	6 (accumulator + currentValue)
네 번째 순회	6	4	3	[1, 2, 3, 4]	10 (accumulator + currentValue)

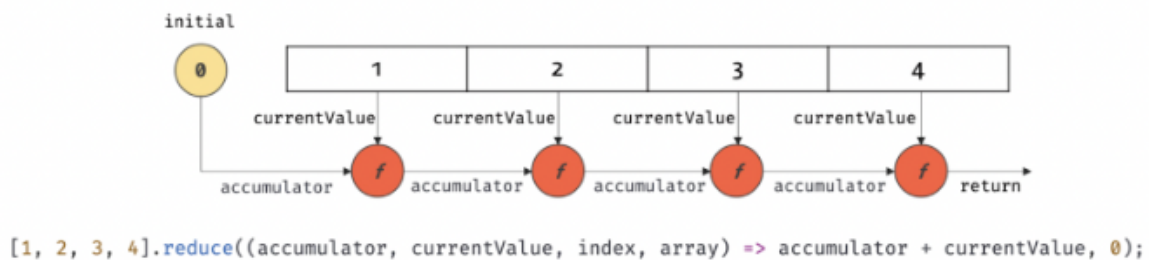


그림 27-11 Array.prototype.reduce

<https://velog.io/@jjinichoi/모던-자바스크립트-Deep-Dive-27장-배열>

객체내의 값 인스턴스 개수 세기

```

const Log = (res) => console.log(res);
const names = ["Alice", "Bob", "Tiff", "Bruce", "Alice"];

const countedNames = names.reduce(function (allNames, name) {

```

```

    //현재값이 누산기 안에 존재하는지 확인 후 카운팅
    if (name in allNames) {
        allNames[name]++;
    } else {
        allNames[name] = 1;
    }
    return allNames;
}, {}));

Log(countedNames)

```

```

▶ {Alice: 2, Bob: 1, Tiff: 1, Bruce: 1}

```

평균 구하기

```

const Log = (res) => console.log(res);
const values = [1, 2, 3, 4, 5, 6];

const average = values.reduce((acc, cur, i, {length}) => {
    Log(length)
    // 마지막 순회가 아니면 누적값을 반환하고 마지막 순회면 누적값으로 평균을
    return i === length - 1 ? (acc + cur) / length : acc + cur;
}, 0);

Log(average);

```

```

6 6
3.5

```

Array.prototype.some

- 자신을 호출한 배열의 요소를 순회하면서 인수로 전달된 콜백함수를 호출

- **some** 메서드는 콜백 함수의 반환값이 단 한번이라도 참이면 **true**, 모두 거짓이면 **false**를 반환
- 배열의 요소 중에 콜백 함수를 통해 정의한 조건을 만족하는 요소가 1개 이상 존재하는지 확인하여 그 결과를 불리언 타입으로 반환
- 빈 배열인 경우 언제나 **false**

매개변수

```
some(function (element, index, array) { /* ... */ }, thisArg)
```

- callbackFn
 - 배열의 각 요소에 대해 실행할 함수
 - element : 처리 중인 현재 요소
 - index : 처리 중인 현재 요소의 인덱스
 - array : filter()를 호출한 배열
- thisArg
 - forEach와 동일

활용

```
const Log = (res) => console.log(res);

Log([2, 5, 8, 1, 4].some((x) => x > 10))
Log([12, 5, 8, 1, 4].some((x) => x > 10))
```

```
false
true
```

Array.prototype.every

- 자신을 호출한 배열의 요소를 순회하면서 인수로 전달된 콜백함수를 호출
- **every** 메서드는 콜백 함수의 반환값이 모두 참이면 **true**, 단 한번이라도 거짓이면 **false**를 반환

- 빈 배열인 경우 언제나 true를 반환

매개변수

- some 과 동일

활용

```
const Log = (res) => console.log(res);

// 배열의 모든 요소가 3보다 큰지 확인
Log([5, 10, 15].every(item => item > 3)); // ->

// 배열의 모든 요소가 10보다 큰지 확인
Log([5, 10, 15].every(item => item > 10)); // ->

// every 메서드를 호출한 배열이 빈 배열인 경우 언제나 true
Log([].every(item => item > 3)); // -> true
```

```
true
false
true
```

Array.prototype.find

- 반환값이 true인 첫 번째 요소를 반환

매개변수

- forEach 과 동일

활용

```
const Log = (res) => console.log(res);
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
```

```
    { id: 3, name: 'Park' }
  ];
```

// id가 2인 첫 번째 요소를 반환한다. find 메서드는 **배열이 아니라 요소를 반환한다.

```
Log(users.find(user => user.id === 2)); // -> {id: 2, name: 'Kim'}
```

//filter는 배열을 반환한다.

```
Log([1, 2, 2, 3].filter(item => item === 2)); // -> [2, 2]
```

// find는 요소를 반환한다.

```
Log([1, 2, 2, 3].find(item => item === 2)); // -> 2
```

```
▶ {id: 2, name: 'Kim'}
▶ (2) [2, 2]
2
```

Array.prototype.findIndex

- 반환값이 true인 첫 번째 요소의 인덱스를 반환

매개변수

- forEach 과 동일

활용

```
const Log = (res) => console.log(res);
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];
```

```
Log(users.find(
1
3
```

```
// id가 2인 요소의 인덱스를 구한다.
Log(users.findIndex(user => user.id === 2));

// name이 'Park'인 요소의 인덱스를 구한다.
Log(users.findIndex(user => user.name === 'P
```

Array.prototype.flatMap

- map 메서드를 통해 생성된 새로운 배열을 평탄화
- 주어진 콜백 함수를 적용한 다음 그 결과를 한 단계씩 평탄화하여 형성된 새 배열을 반환
- 단 1단계만 평탄화

매개변수

- forEach 과 동일

활용

```
const Log = (res) => console.log(res);
const arr = ['hello', 'world'];

// flatMap은 1단계만 평탄화한다.
Log(arr.flatMap((str, index) => [index, [str, str.length]]));

// 평탄화 깊이를 지정해야 하면 flatMap 메서드를 사용하지 말고 map 메서드
Log(arr.map((str, index) => [index, [str, str.length]]).flat(
```



```
▼ (4) [0, Array(2), 1, Array(2)] ⓘ  
  0: 0  
  ▶ 1: (2) ['hello', 5]  
    2: 1  
  ▶ 3: (2) ['world', 5]  
    length: 4  
  ▶ [[Prototype]]: Array(0)  
▶ (6) [0, 'hello', 5, 1, 'world', 5]
```